

Jcc, a native Java compiler

Ronald Veldema
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Jul 31, 1998

Abstract

This thesis describes the design and implementation of Jcc, an optimizing native Java compiler. The system translates Java source code directly to native executables. The system is almost self contained using its own parser generator (Jade) and frontend and backend. The thesis also describes the novel implementation of RMI and the facilities to dynamically load Java byte code into a running executable. The Java runtime system is currently built upon Panda[4], a portable platform to support parallel programming languages. The system runs on both Sparc/Solaris and x86 processors running Linux 2.x.x or BSD/OS 3.0 by BSDI.

Keywords: Java, Bytecode, Offline native compilers, parser generators, native interface.

Contents

1	Introduction	4
1.1	Java	4
1.2	Previous Work	5
2	Using Jade to create the parser	5
2.1	Introduction to Jade	5
2.2	Jade generated parse tree	7
2.3	Creating the parse tree	10
2.4	Scopes and symbol tables	11
3	Object layout in the generated executable	13
3.1	Introduction	13
3.2	Data structures	16
3.3	Meta classes	18
4	Code generation using Gasm	20
4.1	Introduction	20
4.2	The Gasm intermediate code	22
4.3	Mapping Gasm to a target machine	25
5	Java specific optimizations	28
5.1	Introduction	28
5.2	Native interface	28
5.3	Optimizations by using class hierarchy information	31
5.4	Exploiting the lack of pointer arithmetic	31
5.5	Helping the runtime system	32
6	Dynamically loading class files	34
6.1	Introduction to class file usage	34
6.2	Dynamically loading the shared library	36
7	Jcc Remote Method Invocation	37
7.1	Java RMI	37
7.2	Compiler support	38
8	Garbage collection	39
8.1	Introduction	39

9 Java exceptions support	40
9.1 Introduction	40
9.2 Incorporating Unix signals	40
9.3 Creating stack traces	42
10 Results	43
11 Conclusion and future work	44
A Gasm public interface	46
References	48
Index	50

1 Introduction

1.1 Java

Java [6] is quickly becoming a widely used programming language and is beginning to be used by the scientific community [2]. It is also being used to create real world applications where speed is an important factor (consider database/web servers, image processing programs and games). Most current Java environments either use an interpreter of Java bytecode or a just-in-time-compiler (JIT) that does the compilation of Java byte code to native code at runtime. Many applications, however, require speed and can thus not afford the performance penalty of using a bytecode interpreter or even a JIT. The performance of a JIT compiler suffers from having to limit compile time, since compilation is done during execution. This means that only low overhead optimizations such as simple peepholing may be performed (if at all). A traditional offline compiler has no such limitations and can thus use all of the traditional techniques for optimization and perhaps some more, since Java lacks certain features that complicate algorithms for code optimization (e.g. pointer arithmetic and true multiple inheritance). Another reason for using a compiler instead of an interpreter/JIT combination is that it allows us to easily experiment with language extensions. One example is the introduction of “remote” classes for parallel/distributed computing [11][14].

This thesis describes the design and implementation of Jcc, a native offline compiler for Java. One restriction that compiling to native code creates, is the potential loss of mobile code. Mobile code in this setting means that we can take the generated executable, put it on a disk, and read it back on another machine with an entirely different architecture and run it. Jcc does not support this “compile once, run anywhere” model, but it does support dynamic class loading.

Since it is our intention that Jcc be a portable compiler an intermediate language is used. The intermediate language chosen is that of a generalized modern computer, so it has a stack, registers and some amount of memory. This kind of intermediate code resembles the low level intermediate code as described in [12]. Many techniques are known for optimizing such an intermediate code, such as loop invariant code motion, peepholing, induction variable analysis and (partial) redundancy elimination. Current optimizations in Jcc’s backend (named Gasm, for generic assembly language) include peepholing on intermediate and target language, method inlining by analyzing the class hierarchy, copy-propagation, dead-store elimination,

jump-to-jump elimination and aggressively replacing stack positions by registers (register promotion).

The several phases a Java file goes through to create an executable are displayed in figure 1. Since the entire Jcc system is very large (approx. 140.000 lines of C), the thesis describes only the most interesting aspects. Section 2 explains the inner workings of Jade, the parser generator used in Jcc. Semantic analysis and parse tree decoration (assigning offsets to variables and temporaries) is not described here, because it is a simple and straight forward process. Section 3 describes how the Java objects are laid out in memory. Section 4 describes how the intermediate language “Gasm” is used and how certain problems compiling Gasm to machine language are tackled. Optimizations performed, including those on Gasm level, are described in section 5. How dynamically loadable code is treated is described in section 6. Next follow two sections on runtime support for remote method invocation and garbage collection. Both are described more fully in [11]. Section 9 describes how exceptions are handled and how Unix signals are incorporated. Finally section 10, shows some of our measurements, giving some idea of the speedup of our compiler to an interpreter or JIT compiler.

1.2 Previous Work

Cacao [9] is a 64 bit JIT compiler, specifically targeting the Digital Alpha processor; Jcc targets both x86 and Sparc processors. Cacao is able to achieve good performance, but is specifically targeting the Digital Alpha processor. Harissa [13] is a mixed compiler and interpreter. Normally, code will be compiled to C where after it is compiled to a native executable. Dynamically loadable code (byte code files received from the network or loaded from disk), are interpreted. Toba [17] is a Java compiler that targets C, but currently lacks such features as AWT and RMI support. Kaffe [22] is a portable mixed interpreter and JIT.

2 Using Jade to create the parser

2.1 Introduction to Jade

Jade is a recursive ascent [16] [3] [7] parser generator. Basically these operate as follows. First an LR(0) state machine is created which is then turned into a LALR(1) state machine. Next, for every state in the LALR(1) machine, a procedure is created implementing the parsing logic for that state. Shifting from one state to the next thus comes down to calling the procedure that

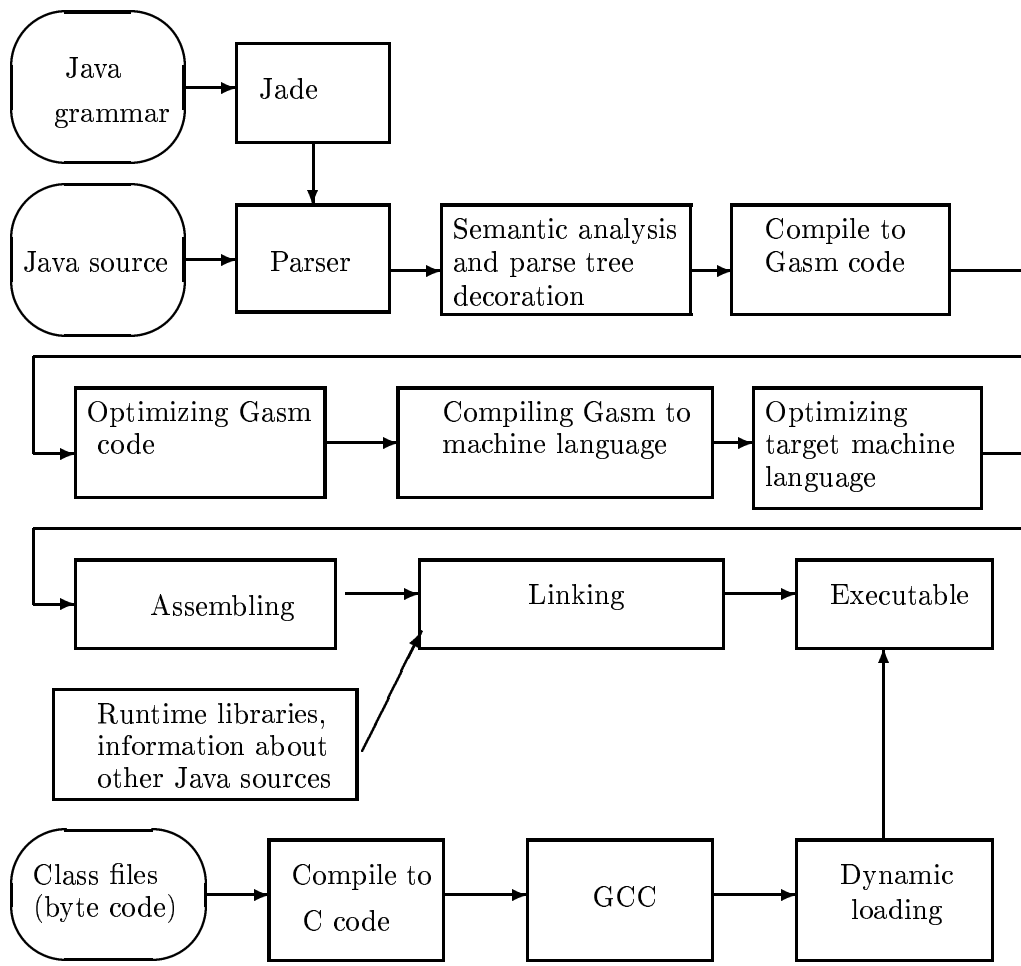


Figure 1: Phases a Java files goes through

implements the next state. When a reduction takes place on rule A at alternative number X with N items, a global variable “KillDepth” is set to N and another global variable “Z” is set to the type-id of A. Then a reducer function is called to call any semantic actions of the user. Next, N return statements are executed, where after parsing can continue using Z in that state/procedure instead of the current tokens type.

There is one major problem with the classic approach described above: parser size [21]. Using the classic recursive ascent parsing method, the generated parser will quickly grow to several megabytes in size. Consider a grammar with a hundred rules, some nullable. This may produce up to a thousand states, where each procedure may be five to six kilobytes on the average. The resulting parser will thus become six megabytes in size.

Jade addresses this problem, not by hard coding parser state in every procedure, but by creating tables to hold state information. Instead of using a procedure per state, we use only one procedure, and give that procedure a pointer to a state structure containing pointers to tables containing the earlier hard coded information (such as error correction tables, shift/reduce decision tables, precedence tables). When entering a new state we call the procedure recursively giving it a pointer to the next state structure. Creating parsing tables also enables us to do some simple compression, again reducing parser size. The generated parser can still be claimed to be recursive ascent, since there are still global variables such as KillDepth used for parsing.

Special features of Jade are: automatic creation of the parse tree, semi automatic creation of symbol tables using restricted formal actions and automatic generation of many utility functions. Symbol table creation is guided by augmenting the grammar with extra keywords such as *%enter_scope*, *%leave_scope* and *%symbol*. A more exhaustive description of Jade and optimizations used to decrease parser size can be found in [21].

2.2 Jade generated parse tree

Jade will, besides generating the LALR(1) recursive ascent parser, instrument the generated parser with code to build the parse tree. The mapping from parse rule to C structure is a simple one. Given a parse rule such as in Figure 2. The structure (parse tree node) in Figure 3 is then generated at parser generation time: The checksum field is used to ensure correctness of a given parse tree. The scope field points to the scope structure active when this rule was reduced (see section 2.4). Line and file tell where in the input file the reduction of this parse rule took place.

The line

```

A -> B '+' D
| D
| IDENTIFIER
;

```

Figure 2: Example grammar rule

```

typedef struct A {
    ParseItemId checksum;
    int line;
    char *file;
    ScopeTree *scope;
    [ struct ParentRule parent_ParentRule; ]*
    union {
        struct {
            struct B *item0;
            /* a terminal '+' */
            struct D *item2;
        } data0;
        struct {
            struct D *item0;
        } data1;
        struct {
            struct Identifier *item0;
        } data2;
    } alternative;
}

```

Figure 3: Structure generated from example grammar rule

```
[ struct ParentRule parent_ParentRule; ]*
```

should be read as a table of pointers to parse tree structures where this rule (A) is used in the grammar. When this structure is instantiated, only one pointer will have a value, the rest will be zero (a node can have only one parent in a tree). The parent pointers are especially useful when having to search down the parse tree to find a particular datastructure. For example, consider the following small example taken from the official Java grammar:

```
MethodDeclaration -> MethodHeader MethodBody Semi_opt;
```

```
MethodHeader -> Modifiersopt Type MethodDeclarator Throws_opt  
| Modifiersopt VOID_TYPE MethodDeclarator Throws_opt  
;
```

```
MethodDeclarator -> IDENTIFIER %symbol '(' FormalParameterList_opt ')' '  
| MethodDeclarator '[' ]'  
;
```

Here MethodDeclarator pointers are inserted into the symbol table since MethodDeclarator declares the method name symbol. When a MethodDeclaration is needed, we have a problem, since lookup for method “foo” will deliver a MethodDeclarator. This problem can be resolved by climbing up the parse tree by following parent pointers. Methods to actually do this tedious job can be generated on demand by augmenting the grammar with:

```
%down_walker GetParentMethodDeclaration :  
MethodDeclarator -> MethodDeclaration;
```

which will generate a routine

```
“MethodDeclaration *  
DownFinderMethodDeclaration_MethodDeclarator(MethodDeclarator *self)”.
```

One problem is that the user might want some extra user defined fields in the generated structures. This is implemented by the user annotating the grammar with

```
%user RuleName { [ type name; ]* }.
```

An example of this grammar rule is shown below:

```
%user MethodDeclaration { int vtable_entry; }.
```

```

typedef struct ParseStack
{
    integer16 item_type;
    union data {
        void *partial_parse_tree;
        double float_constant;
        long long int_constant;
        int char_constant;
        char *string_constant;
        TTokenNode *token;
    } data;
} ParseStack;

```

Figure 4: The parse stack structure

The field “int vtable_entry” will then be entered into the struct belonging to MethodDeclaration. The user must take care not to name a user defined field “file”, “line” or “alternative” since these are already taken. The “C” compiler will, however, complain if this is the case.

Jade also has facilities to create routines to compare and create parse trees and to do search up and down the parse tree. When, for example, building a compiler for a strongly typed language such as Pascal, the type checking facilities can be generated completely.

2.3 Creating the parse tree

When a reduction takes place on the first alternative of the example grammar rule from Figure 2, a new “A” structure is created. The scope field is filled in with the global current_scope field, the item0 and item2 pointers are taken from the parse stack and inserted into the new A structure. After filling in these fields each of these gets their parent_A field set to the newly created structure. Next the top of the parse stack is decremented by the number of items in the reducing alternative (three in the example) and the new A structure is pushed onto the parse stack.

On a reduction, sub-tree pointers must be inserted in the newly created structure. This is implemented using the parse stack. The parse stack is implemented as a fixed size array of ParseStack structures (see Figure 4).

When a shift on a token type is executed the top of the parse stack

is filled in (which fields are affected depends on the type of the current token) and the `top_of_stack` pointer is incremented. On a reduce of length `N` (the number of terminals and non terminals in the alternative) the subtree pointers are taken from the first `N` `ParseStack` items from the top of the parse stack. The top of stack is decremented by `N` and a new `ParseStack` item is pushed on the stack, its `partial_parse_tree` item containing a pointer to the new partial subtree.

For example, when reducing “*B '+' C*”, the top of the stack supposedly contains a ‘*C*’ pointer in its `partial_parse_tree` field. Two items down in the stack there is a `ParseItem` structure that has the ‘*B*’ pointer in its `partial_parse_tree` field. Now a new `A` structure can be created and set up correctly. The top of the stack is decreased by three items and a new `ParseStack` item is pushed with its `partial_parse_tree` pointing to the new ‘*A*’ partial parse tree.

2.4 Scopes and symbol tables

Since Jade is aware of what identifiers are (a special predefined token type), and where they occur in the grammar, two opportunities present themselves. One is to automatically register identifiers in the right scopes combined with a pointer to the parse tree rule struct they occurred in. The other is the opportunity to do a little error checking by checking for doubly declared identifiers either in local or global scope. Since Jade has no information about the semantics of a given grammar file, the user has to tell Jade where scopes begin and end, which identifiers are declarations and which are mere uses of identifiers. This is achieved by annotating the grammar with the keywords “*%enter_scope*”, “*%leave_scope*”, “*%symbol*”, “*%single_local_symbol*” and “*%single_global_symbol*”. “*%single_local_symbol*” and “*%single_global_symbol*”, before entering the symbol into the current scope, check if the declared symbol is not already declared by searching either the local scope or all scopes from the current scope down to the global scope. If it is found an error is printed.

%enter_scope and *%leave_scope* are implemented as follows. When a shift is about to be executed, a check is made to see if the state about to be entered has the ‘`new_scope`’ bit set. If so, a new scope structure (figure 5) is created, it will have the parent scope pointer set to point to the old scope structure and the global `current_scope` pointer is set to point to the new scope. A scope is exited when a next state has its ‘`exit_scope`’ bit set. The `current_scope` pointer is then set to point to the `parent_scope` of `current_scope`.

```

typedef struct ScopeTree
{
    // name of scope (optional)
    char *name;
    // head/tail sub-scope list
    struct TScopeTree *head_sub_scope, *tail_sub_scope;
    // next/previous sub scope
    struct TScopeTree *Next, *Previous;
    // parent scope
    struct TScopeTree *parent;
    // table of symbols in this scope
    TSymbolTable sym_tab;
} ScopeTree;

```

Figure 5: Scope tree node

```

MethodDeclarator ->
    IDENTIFIER %symbol '(' %enter_scope FormalParameterList_opt ')'
| MethodDeclarator '[' ']'
;

```

Figure 6: Example grammar rule demonstrating %symbol

One problem with this approach is when in one parse rule there is both a %enter_scope and a %symbol. In Figure 6, this problem is illustrated (taken from [6]). Since Jade generates recursive ascent LALR(1) parsers a symbol can only be safely added to a scope at reduction time. A new scope, however, can only be created when shifting because entering a new scope at reduce time would be too late and too complicated to correct afterwards. For example, if in Figure 6, in FormalParameterList_opt symbols are added to the symbol table but %enter_scope would be implemented at reduction time, the symbols exported in FormalParameterList_opt would be entered into the wrong scope. This is, because FormalParameterList_opt would be reduced before MethodDeclarator would. The solution is to create scopes when shifting to a new state and at reduction either adding a symbol to the parent's scope or to the current scope depending if in the rule alternative the %symbol came before or after the %enter_scope.

3 Object layout in the generated executable

3.1 Introduction

The way objects are laid out in memory in the final executable greatly impacts performance. In Java the situation is complicated by the possibility of introspection of objects and class hierarchies. Efficient garbage collection and remote method invocation also have their impact on object layout. Yet another factor to be taken into consideration is that the interface between the Jcc runtime (written in C) and the compiled Java code should be easy to use and fast. To accomplish these goals, at compile time, for every Java class an equivalent C structure is emitted. The runtime system can thus directly operate on these structures instead of querying the meta class. The Meta class however still contains information such as field and method information tables for compatibility and supporting dynamic class loading.

One of the goals of Jcc is to support efficient RPCs (less than 40 microseconds on a Myrinet network[11]). To support this goal, every class or interface type has a type-id and a table of constructors, which can be used to remotely create and initialize an object given a type-id. In Java, all method invocations are virtual by default (see [5]), and non virtual only for methods that are marked static and final. Calls may also be made non virtual for methods from final classes. In other words dynamic binding is the default in Java whereas static binding is the default in C++.

Virtual method calls in Jcc are implemented using virtual function tables. A virtual function table (vtable, from here on) is a table of function pointers for the methods of that class. In Jcc, this is extended to include per object constant data in the vtable header, such as meta class pointers. Some per object constant data is however allocated in the object itself to improve performance.

Every class instance has a pointer to its vtable. When one class inherits from another a copy of the vtable is made at compile time. When a method is overwritten by the inheriting class, that function pointer entry in the table is overwritten (at compile time). When a virtual function call is made, instead of directly calling the method, the vtable is indexed to find the correct method pointer. This way the most derived method is always called for that class instance, instead of the compile time visible method. Behavior of a remote procedure call and a local procedure call should ideally be the same, therefore a virtual function table of unmarshall functions is maintained in the shadow vtable. Every method from a remote class thus has an entry here that points to the unmarshaller for that method.

Given the following class:

```
class Monkey {
    int value1;
    Monkey m;
    int value2;
    String s = "Banana";
    int value3;
}
```

The following tables are produced:

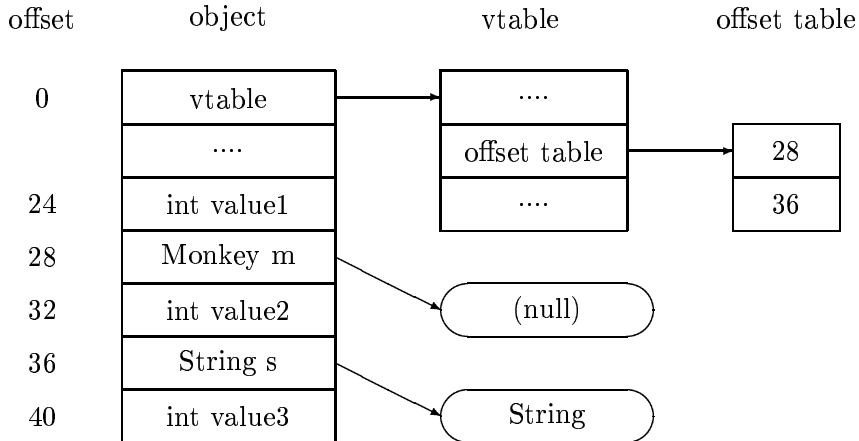


Figure 7: Object offset table

To support garbage collection, without scanning the entire object, the garbage collector needs to know which variables in the object are object pointers. This is implemented in Jcc by maintaining a table of offsets in the objects to pointers (see Figure 7). A pointer to the offset table is inserted into the header of the vtable. In Java an array is also an object which in turn may contain different object pointers as its data. This implies that the garbage collector will always need to scan the entire array. If an array is primitive however (meaning that it contains only primitive data types) the garbage collector can skip its contents without any further examination. The same applies to serialization. When a primitive object is encountered during serialization, it can be copied onto the output in one write operation without further checking. This is supported by adding a bit to the flags field of every object and array, indicating whether the object is primitive

A and B vtable combined	A's data	B's data	A and C vtable combined	C's data
-------------------------------	----------	----------	-------------------------------	----------

```

class A { int data1; }
class C { int data2; }
class B : public A, C { int data3; }

```

Figure 8: Multiple inheritance in C++

(contains no pointers).

In Java, multiple inheritance is supported by the use of interfaces. An Interface is a class which is not allowed to carry any state. This stateless feature allows us to do some optimizations. For garbage collection and serialization purposes it is desirable that object pointers always point to the start of the object and not to the middle of an object. In C++ for example multiple inheritance is implemented using some pointer arithmetic. When casting a multiple inherited object to one of its base classes the pointer may be reset to point to the base object's vtable pointer as shown in Figure 8. Here when casting from a B pointer to a C pointer, the pointer will be reset to point to the middle of the object.

In Jcc multiple inheritance of interfaces is implemented by using fat pointers. An interface pointer thus consists of the object pointer and a pointer to a virtual function table for that interface (the interface view). This way, object pointers still point to the start of the object at the cost of the larger pointer size. When a virtual function call is executed, instead of looking up the function pointer in the normal vtable (the vtable pointer located in the first few bytes of the object) the function pointer is taken from the interface view.

When a class implements an interface, the interface view pointer is appended to the end of the object. To execute a cast from a class to an interface, the correct view pointer on that interface is extracted from the object and appended to the object pointer to form a fat pointer. When casting an interface pointer back to a class pointer, the extra bytes of the interface view pointer are simply thrown away.

<i>javaVTable *</i>	<i>vtable</i>
<i>java_mutex_t *</i>	<i>object_lock</i>
<i>int</i>	<i>type_id</i>
<i>int</i>	<i>flags</i>
<i>int</i>	<i>object_size</i>

Figure 9: Object header

```
Object o = new Object();
synchronized (o) {
    System.out.println("in a synchronized block, only one thread can enter this block;");
}
figure 4.
```

Figure 10: Example synchronized statement

3.2 Data structures

Every object has a standard object header. This includes normal objects, arrays, meta classes and remote object stubs. The equivalent “C” standard object header is displayed in Figure 9.

The *vtable* field points to a method table and other per object constant data (Figure 12). The object lock field is required because, in Java, every object can be used as a lock (by using it in a synchronized statement or by declaring a method in it synchronized). See for example the Java code fragment in Figure 10.

The *type_id* field is a program wide unique number used for implementing RPC and dynamic type checking. When given a type-id, the meta class for that class can be found. Using the meta class new instances can be created and properties queried. The *flags* field contains bits for checking properties of an object. There are bits defined for cloneable, serializable, runnable(=thread), array, primitive, static, remote and more. The Cloneable bit is used for testing if a class is an implementor of interface Cloneable. When the clone method is invoked and this bit is not set a CloneNotSupportedException is thrown. The Runnable bit is set when this object is a descendent of interface Runnable. This bit is mainly used by the garbage collector [11]. The Array bit is set when this object is an array. The Primitive bit is set when no references occur in the object or its parent classes. The remote object bit is set when this object is a remote object. If this bit is set the garbage collector and serializer must take extra care when handling

<i>int</i>	<i>interface_id;</i>
<i>int</i>	<i>offset;</i>

Figure 11: *InterfaceOffset* structure

<i>javaParentTab * []</i>	<i>parent_table</i>
<i>int *</i>	<i>offset_table</i>
<i>struct javaClass *</i>	<i>class_class</i>
<i>javaVTableEntry *</i>	<i>shadow_vtable</i>
<i>javaInterfaceOffset *</i>	<i>view_offset_table</i>
<i>java_functionPointer []</i>	<i>entries</i>

Figure 12: *VTable* structure

this object type. The garbage collector is not allowed to directly throw such an object away because there may be remote references to it. The serializer must take care to tell the garbage collector about new remote objects after deserialization. The object size field is used for serialisation purposes (fast copying of objects and arrays, for example). The static bit is set when the object has been statically allocated and can thus not be removed. An example of this, is when the object has been allocated as a global structure. No 'new' will thus have been performed on the object and the garbage collector is thus not allowed to 'free' it. An example object where this bit is set, is a string constant that may be statically allocated in a read-only segment.

The *javaVTable* structure of per object constant data is displayed in Figure 12. The *parent_table* field is a pointer to a zero terminated array of type-id's of parent classes and parent interfaces. Most of the tables generated by *jcc* are zero terminated. The offset table indicates which variables in the object and its parent classes are object pointers. This limits the amount of search the garbage collector needs to do per object. The field *class_class* points to the meta class belonging to the object instance. The shadow vtable is a table of function pointers to unmarshallers. For every function in a remote class method table there is a function pointer here that unmarshalls parameters from an input stream and calls the real method. The *view_offset_table* field points to a array of *javaInterfaceOffsets* structures. A *javaInterfaceOffset* structure (see Figure 11) is a pair of *interface-id* and *interface view offset* in the object implementing the interface. When a cast is executed from an object to an interface and we can not statically determine the offset the interface view pointer has inside the object, search has to be performed. When the search fails a *ClassCastException* is thrown.

<i>javaObject</i>	<i>base</i>
<i>int</i>	<i>element_count</i>
<i>int</i>	<i>element_size</i>
<i>ArrayElementType []</i>	<i>data</i>

Figure 13: Array layout

In Java an array also derives from Object and is thus an object. This practically means that it will have the same layout in memory as Object but with some extra fields. The layout of an array in memory is shown in Figure 13.

The field 'base' is allocated to make an array compatible with Object shown in Figure 9. The *element_count* and *element_size* fields speak for themselves. The *element_count* field is required for array bound checking. The *element_size* field is useful for implementing arraycopy (from `java.lang.System`) efficiently (amongst others). The *ArrayElementType* (of course) differs per array type.

3.3 Meta classes

In Java every class has a companion meta class, simply named 'Class'. Given a meta class, one can ask it to create a new instance of the describing class (`Class.newInstance`), ask it what the meta class is of the describing super class and some other less useful functions for a static compiler (e.g., to get its class loader and query for class methods and fields).

One interesting feature of a meta class is that we can statically ask it to find the meta class for an object given the class name. To support this feature, tables are maintained mapping class names to meta class references. To enhance upon this functionality, the type-id of the class the meta class belongs to is also added to this table. When, for example, the RMI implementation on one machine receives a request to create a class over a network, the meta class of the object can be looked up in this table and it will create a new object. This does, however, require that all class-id's are unique. This is ensured by first calculating a new type-id based upon the object's name and package it was declared in. Then a search is made in the table of all objects used in this project to see if the type-id has already been used. Given the randomizing strength of the hash function upon object names, type-id clashes have so far not been observed. Many other techniques to achieve this effect are available, see for example [19].

The complete meta class information maintained is displayed in Fig-

<i>javaObject</i>	<i>parent</i>
<i>int</i>	<i>is_interface</i>
<i>struct javaClass *</i>	<i>parent_class</i>
<i>int</i>	<i>size</i>
<i>void (*)()</i>	<i>default_ctor</i>
<i>javaString *</i>	<i>class_name</i>
<i>int</i>	<i>object_type_id</i>
<i>javaVTable *</i>	<i>object_vtable</i>
<i>int</i>	<i>object_flags</i>
<i>int</i>	<i>methods</i>
<i>javaMethodDescriptor *</i>	<i>method_descriptors</i>
<i>javaFieldDescriptor *</i>	<i>field_descriptors</i>
<i>java_constructor[]</i>	<i>constructors</i>

Figure 14: Java meta class layout (class Class)

ure 14. The meta class is (like any other class) a descendent from java.lang.Object, which explains the parent field. The is_interface field indicates that this meta class describes an interface or a normal class.

As a side note, the Java language specification is ambiguous to what this field means. Interfaces can't exist on themselves, they must be implemented by a class to be instantiated, therefore the dynamic type of the interface pointer ought to be that of the implementing class. So asking an interface reference what its meta class is *should* deliver that of the implementing class.

The parent_class field points to the meta class of the base class of the class this meta class describes. The size field is set to the size of the class this meta class describes. The default_ctor field points to the generated default constructor of this class. This is necessary when a class instance is needed but no user code should be invoked on creation. Class_name is set to point to a Java string object (instance of java.lang.String) that contains the name of the describing class. Object_type_id, object_vtable and object_flags fields are set to the type-id, vtable and bit fields of the describing class respectively. object_size, object_vtable and object_type_id fields are all that is required to create a valid Java object. Consider, for example, when deserialization needs to deserialize an array of objects efficiently. It may create, and initialize, the objects to be inserted into the array quickly by doing the lookup of these fields only once. The method_descriptors field points to an array of javaMethodDescriptor structures. These contain method names (both symbolic, mangled, and as a valid method descriptor string as prescribed by SUN). It also contains the starting and ending addresses in the text segment. The latter is used, for example, by the exception handling code (see

<i>void *</i>	<i>start_func;</i>
<i>char *</i>	<i>mangled_name;</i>
<i>char *</i>	<i>symbolic_name;</i>
<i>char *</i>	<i>descriptor;</i>
<i>char *</i>	<i>name;</i>
<i>int</i>	<i>vtable_offset;</i>
<i>int</i>	<i>param_count;</i>
<i>void *</i>	<i>end_func</i>

Figure 15: Method descriptor

<i>char *</i>	<i>name;</i>
<i>int</i>	<i>offset;</i>
<i>int</i>	<i>flags;</i>
<i>int</i>	<i>type_id;</i>

Figure 16: Field descriptor

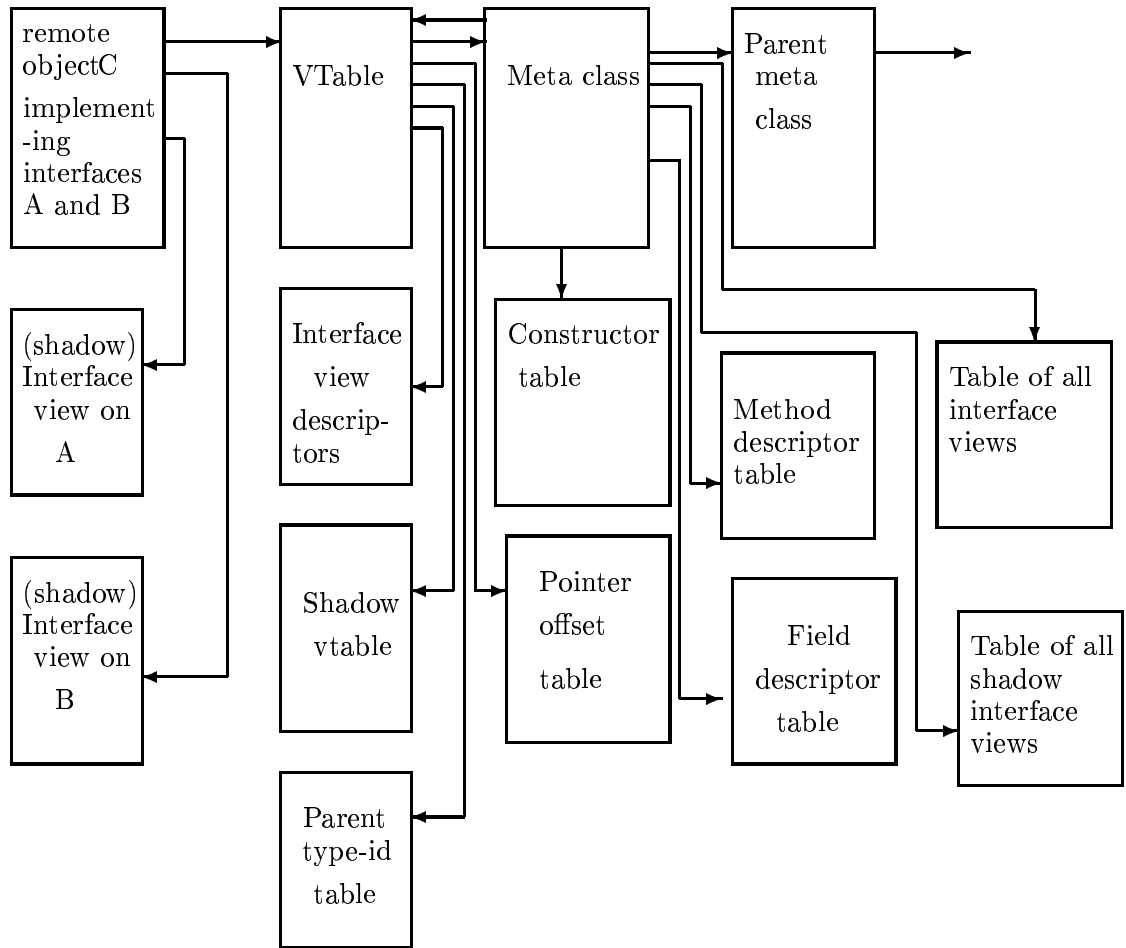
section 9, on exception handling). The actual layout of a `javaMethodDescriptor` is displayed in Figure 16. The `field_descriptors` field points to an array of `javaFieldDescriptor` structures. When compiling a class file to C this information can be used to do a minimal amount of error checking. If, for example, a method declared in a class file uses a field from a compiled class we can check in advance if the field is actually present and what offset it is at. The actual layout of a `javaFieldDescriptor` is displayed in Figure 16. The `constructors` field contains an array of function pointers to constructors defined by the described class. When an RMI is executed the receiving end of the call receives as part of the 'new' message an index into this table to be able to call the right constructor from the unmarshaller [11].

To further illustrate the object layout, a visual representation is shown in Figure 17 of a remote class implementing two interfaces A and B.

4 Code generation using Gasm

4.1 Introduction

Compilers require an intermediate language to achieve good portability and performance. Many such intermediate languages have been proposed in the past, all differing in approach and effectiveness for optimization. Performance also depends on the effort invested in classical optimizations, de-



```

interface A { void foo(); }
interface B { void bar(); }
remote class C implements A, B {
void foo() {}
void bar() {}
}

```

Figure 17: A remote class implementing two interfaces

scribed for example in [12].

Completely stack oriented intermediate languages such as EM[20], are generally easy to use, but are difficult to port efficiently to different architectures. For example, many RISC architectures are less efficient in handling stack operations but instead rely heavily on register usage to store variables and temporaries. A simple, naive way, of implementing a push and pop operation on a machine that does not have these operations is to emulate their effects by designating one register as a working stack pointer and manipulate memory using register indirection and pointer arithmetic. This will however impair performance as this register can now no longer be used for other purposes such as optimization. Another side effect is increased memory usage.

A purely register based approach, is generally better suited for optimization. An example of the latter category is RTL (register transfer language) used in for example GCC. GCC also demonstrates the high portability achievable by using a RTL given the large number of machines GCC has been ported to.

4.2 The Gasm intermediate code

Gasm takes the middle road between using a completely stack based approach and using a registers only approach by supplying both a limited number of registers and a stack complete with push and pop instructions. In terms of [12], it would be an LIR, a low level intermediate language.

Gasm includes instructions such as move, add, and, or, sub, mul, div, xor, or, compare, push and pop. A complete listing of available Gasm instructions is included in appendix A. Move, add, sub etc., are all two operand instructions. A piece of sample Gasm code to compute $A = (B * C) + D$ is shown below

```
move R0, B  
move R1, C  
multiply R0, R1  
add R0, D  
move A, R0
```

Normal arithmetic and data movement instructions generally operate on registers instead of on the stack. To support conditional code execution and looping there is an invisible flags register that can be modified using the compare instruction on floating point or integer numbers. Its state can then be tested using `Jump{Bigger,Smaller,BiggerEqual, SmallerEqual}`.

Floating point support is implemented through a strict stack machine model. Using `float` and `double_push_literal`, and `{double, float}_{add, sub, div, mul, mod}`. There are also instructions to convert the top of the stack to a `{float, double, int}`.

Gasm currently has five front end visible registers as wide as the registers of the target machine. Internally however, Gasm has as many registers as the machine targeted for. The extra available registers are used for optimizations (pulling locals and temporaries into registers). This last property ensures that there will be a minimum of code duplication in register allocation code since this can now mostly be done on the intermediate machine instead of on the target intermediate directly. The user visible registers are furthermore divided into four general registers and a base pointer. There is also the implicit use of the stack pointer in the push and pop instructions, but because Gasm has several meta statements that indicate what the relative value of the stack pointer should be at every point in the program the stack pointer can be omitted.

Take for example, the piece of Java code and its corresponding (unoptimized) Gasm code shown below:

```

    int a = 3, b;
    b = a < 3 ? 4 : 5;
    {allocate a to [FP-4], and b to [FP-8],
     where FP stands for "stack frame pointer"}
    move [FP-4], 3
    move R0, [FP-4]
    compare R0, 3
    jump lower 1
    push 5
    jump 2
1: push 4
2: pop [FP-8]

```

If we, at compile time, use a single global variable, RSP, to indicate what the FP relative stack pointer should be at every statement, the conditional push in the example poses a problem. Just after the compare, RSP will be zero to indicate that SP equals FP. Just after both push statements, and just before the pop statement, the values of RSP should be equal. If we just linearly scan over the code, adding to RSP at every push, subtracting from RSP at every pop and not looking at conditional code execution, RSP will be twice the word size just before the pop statement (seen 2 push statements).

The solution taken in Gasm is to provide SaveStackSize and LoadStackSize meta instructions. They need to be inserted before the jump lower and after label 1 respectively. Inserting these meta statements is nearly trivial to the frontend and only rarely needed, so most push and pop statements can be resolved without these. The only places where they actually need to be inserted is around the conditional push statements as shown above. A call statement will automatically decrease RSP with the number of arguments in the call.

The results of all our efforts is that we will have rewritten all SP relative memory references (implicitly in the push and pop) to FP relative references. The example code thus becomes:

```
move [FP -4], 3
move R0, [FP -4]
compare R0, 3
jump lower 1
move [FP -12], 5
jump 2
1: move [FP -12], 4
2: move [FP -8], [FP -12]
```

Register promotion may then pull FP relative references into registers, possibly removing all memory references in this example.

To support proper compilation to machine language and to support good optimization and to enhance portability, high level details may have to be conveyed to the optimizer and the backend. In Gasm this is implemented by using meta commands. Meta commands include StartFunction "X", EndFunction, BeginBlock(block-type), EndBlock and DeclareUnOptimisable(stack-offset). StartFunction and EndFunction define the bounds of a function. DeclareUnOptimisable(stack offset) declares that a local variable at that offset must not be touched by the optimizer. This allows the proper implementation of volatile variables. Volatile variables are important for Java given its multithreaded nature and to help garbage collection. The garbage collector can now be sure that at least somewhere a variable will be referenced, either on the stack or in the heap. Other functions are supplied to convey line and file positions in the original source file. Gasm may then store this information for debuggers such as GDB.

Creating a compiler using Gasm is straight forward. First one inserts a call to OutputProgramHeader(program_name) at initialization and a call to OutputProgramTrailer at the end of the compiler. Next when traversing

functions and constructors, calls to `FunctionHeader(function_name, max_locals)` and `FunctionTrailer()` are made. When compiling a statement block, calls to `StartBlock(block-type)` and `EndBlock` are made. `StartBlock` is given a block type parameter to indicate that this statement block is for example a “loop”, “while”, “do” or statement block. An example Gasm program using start and end block meta statements is displayed in figure 18. Next normal Gasm statements can be used such as `PushR0`, `MultiplyR0(10)` etc., to actually do the work. Next when the function result is ready to be returned it can be pop’ed into a float or integer return register. The function that did the call to this function can then retrieve the function result by doing a corresponding push integer or float return register.

Global data structures can be allocated in several sections according to their needs. For example there is a read only section well suited for constant strings, virtual method tables and others. Next there is a data segment for allocating at runtime changeable structures and values. These include global arrays and structures and global numerical type values. Allocating a single global integer initialized to zero can be expressed as:

```
AllocGlobalDataHeader("variable_name", DataSeg);  
ItemAllocNumber(0);
```

Debugger support is available through the functions `OutputDebugHeader`, `StartDebugBlock` and `EndDebugBlock`. Just after `OutputProgramTrailer` a call to `OutputDebugHeader` is made. After calls to `StartBlock` and `EndBlock`, calls to `StartDebugBlock` and `EndDebugBlock` are made. Debugging information can then be added to the stack frame in that block by adding calls to `AddDebugInfoParameter(name, offset, debugging_id)` and `DeclareDebugLocalVariable(name, offset, debugging_id)`. A new debugging type-id can be obtained by calling `GetNewDebuggingId()` and using that number to instantiate a debug-type using `OutputCardinalTypeDebug` and `OutputEnumTypeDebug`. More will be added as Gasm matures. Line and file number information can be added to the output by inserting calls to `NewAsmFileLocation` at a few locations in the compiler. Suggested places are before every statement and before every `StartBlock`. This allows GDB to find source locations for statements when using GDB list-source command.

4.3 Mapping Gasm to a target machine

Porting Gasm requires making a mapping of a virtual machine with a stack and a minimum of five registers to the target machine. This may pose problems when the target machine does not have one of the basic instructions

Given the Java method:

```
int foo(int x) {
    int k = x;
    for (int i = 0 ; i < 10 ; i++) {
        k += i;
    }
    return k;
}
```

Jcc will create the following Gasm code:

```
FunctionHeader "foo", 8 bytes for locals
move R0, [FP + 0]
move [FP -4], R0
move [FP -8], 0
StartBlock: loop
2:
compare [FP -8], 10
jump bigger equal 1
move R0, [FP -8]
add R0, [FP -4]
move [FP -4], R0
add [FP -8], 1
jump 2
1:
EndBlock: loop
load int return [FP -4]
FunctionTrailer
```

Figure 18: Sample translation of Java method to Gasm

available in Gasm. Consider, for example, the Sparc RISC processor. It does not have a normal working stack as used in Gasm, which poses a problem when creating a mapping for the push and pop instructions. There are many possible approaches to this problem. One can, for example, use a free register as a new stack pointer, or one can hardcode all push and pop instructions to fixed stack addresses. The latter is the preferable method since it exposes new optimization opportunities to the optimizer (it can now load a heavily used stack position into a register). This behavior currently has to be implemented by hand, as an ordinary code generator generator would most probably have found the previous mapping.

Another reason against using a code generator generator, besides efficiency, is needed effort. Creating a complete machine descriptor for a Sparc processor is as large a job as creating the backend by hand. The GCC machine descriptor for the Sparc architecture is approx. 190 K of code. There are also some additional files to do some patching to make it conformant to the Sparc ABI (Application Binary Interface). In total the Sparc dependent directory is approx. 620 K for GCC. The Gasm Sparc dependent directory is approx. 258K total.

Creating a backend for a x86 type processor is nearly trivial since all Gasm instructions are available and mostly in the same format. Mapping target machine registers to Gasm registers is also a nearly trivial job since both the Sparc and the x86 have more than four general registers (x86 has six namely eax, ebx, ecx, edx and esi, edi). The first four are mapped to the four general registers, the leftover registers can be used for optimization and patching. On a Sparc processor, a plethora of mappings is possible and still leaving a good number of registers for optimization. In the current implementation, a mapping is made from Gasm R0-R4 to Sparc %g1-%g4. The Gasm stack frame pointer is mapped to %FP .

To be completely Sparc ABI compliant, register windows need to be used for parameters passing. This, however, is now an easy thing to do, simply divide the offset to a parameter by the wordsize to get X. If X is smaller or equal to the number of registers available for parameter passing, Gasm will use %ix, otherwise, Gasm will resort to using the normal hardware stack indexed from %FP + constant + offset. %lx and %ox are left for optimizations. If a call is executed the first N parameters are copied from the working stack to the register window argument registers %o0-%o7 (In Gasm a call specifies how many parameters it is using). Any access parameters are left on the runtime stack (as the Sparc ABI prescribes).

```

class A {}
class B {}
class C {
static void foo() {
    A a = new A();
    Object o = a;
    B b = (B) o; // attempt to cast a A to a B
    }
}

```

Figure 19: Example code throwing a class cast exception

5 Java specific optimizations

5.1 Introduction

Java has some unique features lacking in Pascal and C/C++ that enable certain optimizations. Such features include the absence of varargs, the impossibility to take the address of a local variable or parameter and pass it around and the absence of pointer arithmetic. This allows us to greatly simplify aliasing analysis and thus to pull more variables into registers.

Java also has some language features that cause a performance hit, for example, the abundance of runtime checks such as cast exceptions and array bounds checking. Cast exceptions occur when casting one class to another where the class to be casted is not a base of the other. An example of this behavior is given in example 19. The last line in `foo()` will throw a `ClassCastException` because `B` is not a base or derived class of class `A`. The Java standard also defines that divide by zero exceptions and null pointer exceptions should be thrown as normal Java exceptions thus catchable by user code. This behavior can be implemented either naively by checking every division and pointer access or by using hardware support. The latter approach has been taken by Jcc and is described in section 9.

Another Java feature that may make Java take a performance hit over, for example C, is that every call is virtual. In some cases the virtual call may be optimized away (see Section 5.3).

5.2 Native interface

To optimize the interface between the Java code compiled by Jcc and the runtime libraries, a different native interface than SUN prescribed JNI (Java

```

class A {
    int a = 5;
    native void foo();
    static void main() {
        foo(new A());
    }
}

```

The native method foo will look like:

```

JNIEXPORT void JNICALL Java_A_foo(JNIEnv *env, jobject obj) {
    jclass cls = (*env)->GetObjectClass(env, obj);
    jfieldID fid;
    jint value;
    fid = (*env)->GetFieldID(env, cls, "a", "Ljint;");
    value = (*env)->GetObjectField(env, obj, fid);
    printf("a = %d", intr);
}

```

Figure 20: Standard Java native interface (JNI)

Native Interface) is used. First of all, all Java primitive types such as double, int and long have been made compatible with their C counterparts. Next for every class a matching C structure is emitted. In JNI, search is performed when looking for a class variable (GetFieldID()). When a variable is found, get and set routines are used to access the variable (GetObjectField()). Consider the example in Figure 20, where we want to print the value of 'a' in the native method foo().

In Jcc, marking a method native will mark it for use as a C style function and use the C calling convention reversing argument evaluation and mangling the method name for that platform (normally there is no name mangling in C but some C compilers prepend an underscore, normal method's names are mangled using the same name mangling as C++ (see [5]).

The downside to using the C calling convention, is that it precludes overriding and overloading of native methods, because C does not mangle function names properly. The advantage is that there is absolutely no overhead in calling a native method. An example of using our C interface is given in Figure 21.

The method Math.cos() called from main is a native method, which is

```

class test {
public native double cos(double a);
public static void main(String arg[]) {
    System.out.println("cos(3.1459) = " + cos(3.1459));
}
}

```

Figure 21: native interface of Jcc

```

class test {
public native int number_of_cpus, current_cpu_number;
public static void main(String arg[]) {
    System.out.println("number of cpu's = " +
        number_of_cpus +
        ", current cpu = " +
        current_cpu_number);
}
}

```

The variables `number_of_cpus`, `current_cpu_number` are thus declared in C as: `extern int number_of_cpus, current_cpu_number;`

Figure 22: example using native static variables

turned into a direct call to the `cos` function from `libc`. On a processor such as the x86 where the math coprocessor has built in instructions for calculating the cosine, sinus, square root etc, the call can be optimized even further by the backend turning it into a single instruction.

To even further tighten the interface between Java and the runtime system, variables can also be marked native to allow access using a C style addressing mode. This way a global variable of the runtime system can be easily accessed from Java as if they were static variables. Handling matters this way, in many cases, saves us a call to the runtime system to get or set the global variable. An example of the use of native variables is demonstrated in example 22.

If using Java still incurs too much overhead, and calling a C function to do the actual work still involves too much call overhead, it may be eliminated using some inline assembly. In addition it may be useful to use inline

assembly in performance critical procedures where performance is essential and Jcc fails at optimizing it satisfactorily. In Jcc the syntax used for inline assembly is:

```
asm("string such as, move $0, %eax")
```

5.3 Optimizations by using class hierarchy information

Java is an object oriented language and thus encourages the use of many small methods and constructors. Many such procedures will only be setting or getting a single variable. Such small procedures should be inlined to achieve good performance.

To be able to inline a function at a call site, Jcc must be able to statically determine the class the method belongs to. If this class is declared final, the virtual method call can be eliminated and a direct (normal) call can be used instead. The same applies when a method is never overridden by a derived class. This is however complicated by the fact that a derived class may be loaded at runtime with a method overriding the original method. Most of the time (in Jcc at least) classes will not be loaded at runtime and we can turn every virtual method call into a normal call when a method is not overridden and Jcc sees that this file will be turned directly into an executable. This behavior can, however, be overwritten by a command line switch to disable this optimization and make the compiled code suitable for use in (shared) libraries and enable the use of dynamically loaded classes.

5.4 Exploiting the lack of pointer arithmetic

Because Java lacks the possibility to take the address of some field or local and also lacks the possibility to do pointer arithmetic, aliasing analysis becomes very simple. Take for example the following piece of C code below:

```
void foo() {  
    int a = 1;  
    int *b = &a;  
    *b = 2;  
    printf("a = %d", a);  
}
```

Here analysis has to be performed to see that 'b' aliases 'a'. If an optimizer erroneously pulls 'a' into a register this example would print '1' instead of the correct '2'. In Java one cannot write something like lines two and three.

```

void foo(int a) {
    int i, j, k;
    // emit meta statement: start if block
    if (a > 0) {
        // emit meta statement: start loop block
        for (i = 0; i < 10 ; i++) {
            a++;
        }
        // emit meta statement: end loop block
    } else {
        j = a;
    }
    // emit meta statement: end if block
    .
    .
}

```

Figure 23: Using block meta statements

In the Gasm backend heuristics are used to calculate the approximate usage frequency of local variables. The most frequently used variables can thus be pulled into registers improving performance. It is thus essential that the heuristics to estimate usage frequency perform well. In Jcc, the compiler frontend tells Gasm where every block starts and ends and what kind of block it is. An example of using block meta statements is displayed in Figure 23. Here 'a' and 'i' are prime candidates for register allocation, since they will get the highest score for being used in a loop. Gasm can see this because the uses of the stack slots for 'i' and 'a' will be enclosed in loop-block-start and loop-block-end meta statements. Inserting such meta statements is almost trivial for a front end and saves the backend the work of creating dominator trees [12].

5.5 Helping the runtime system

Another place where the absence of pointer arithmetic helps is in the garbage collector and the serializer. It can now speed up scanning of stacks for references, because a pointer always points to the start of an object and never somewhere in the middle. Also, the object serializer never has to

```

class A implements I {}
class test {
    void foo() {
        A a = new A(); // line 1
        I i1 = a; // line 2
        Object o = a; // line 3
        I i2 = (I) o; // line 4
        I i3 = (I) new Object(); // line 5
    }
}

```

Figure 24: Cast exceptions

search for the start of a given object. Multiple inheritance using interfaces poses a problem in this respect. In C++, multiple inheritance is supported by appending the data for every multiply inherited class to the end of the object. For example, figure 8 displays a vtable of class 'C'. When casting from class B to class C, the pointer is moved to point from the beginning of the object to the vtable of A and C combined. This is an undesirable property and Jcc works around this by using 64 bit pointers instead. In Jcc an interface pointer is a structure of {object pointer, vtable pointer}. Inside an object the internal layout is still the same as for the C++ case. When casting from an object to an interface, the interface 'view' is placed in front of the pointer to form a 64 bit pointer. When it is possible at compile time to see what the right vtable offset is, the vtable will be pulled from the object itself directly. When it is impossible to statically determine the offset of the correct interface view, the meta class of the object (see chapter 3, internal object structure) is searched for the interface vtable given the interface typeid. If a matching interface view is not found a ClassCastException is thrown. This occurs in the example in figure 24. At line 2 the class information is sufficient to lookup the interface I view on A. At line 3 all type information is basically lost, since in Java all objects are descendent of Object. At line 4 therefore the meta class of A must be consulted where the interface view of A on I is located. At line 5 an illegal cast is attempted which will throw a bad cast exception. This is because no interface view information about interface 'I' will be found in the Object meta class.

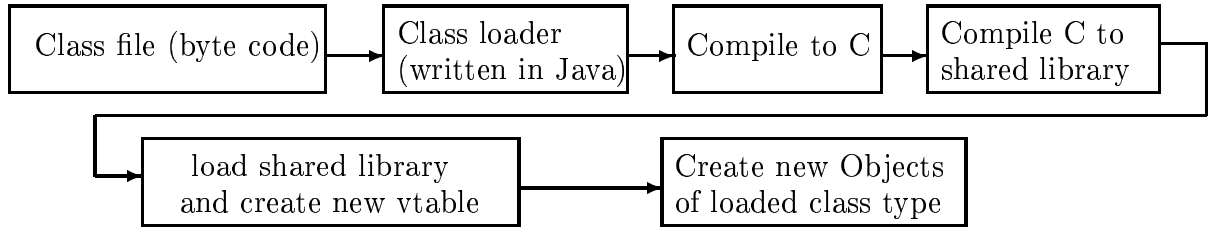


Figure 25: The phases executed when dynamically loading a class file

6 Dynamically loading class files

6.1 Introduction to class file usage

In Java, a class file may be dynamically loaded into a running program and its methods called. This poses problems for a statically compiled program. Matters are complicated even more by the introspection/reflection abilities of Java. To prevent that code from dynamically loaded classes runs much slower than normal compiled code, the byte code is translated on the fly to C. The C code is compiled to object code to form a shared library by invoking a shell command to run GCC. The complete pipeline for dynamically compiling and loading a class file is shown in figure 25.

In a class file, byte code is declared for an abstract stack machine. It contains instructions such as *LDC*, *LOADI* and *IADD*, for pushing a constant, pushing a local variable and adding the top two elements from the stack and pushing the result, respectively. Every byte code is next translated into a call to a macro to do the actual work. The working stack for the instructions is allocated on the procedure stack. When for example the following snippet of Java code is compiled

```

void start() {
    int a = 1;
    int b = 2;
    int c = a * b;
    int d = c;
    return d;
}
  
```

The following byte code will be produced:

```

0 iconst_1
  
```

```

1 istore_1
2 iconst_2
3 istore_2
4 iload_1
5 iload_2
6 imul
7 istore_3
8 iload_3
9 istore 4
11 iload 4
13 ireturn

```

After loading the byte code, the following C code will be produced.

```

int method_test_start0(javaObject *self) {
    int ret_val = 0, sp = 0, call_ret_val = 0;
    TOperandStack operand[MAX_OPERAND_STACK];
    TOperandStack locals[MAX_LOCALS];
    javaObject *call_exception = 0;
    PUSH_CONSTANT(1);
    STORE_LOCAL_INT(1);
    PUSH_CONSTANT(2);
    STORE_LOCAL_INT(2);
    PUSH_LOCAL_INT(1);
    PUSH_LOCAL_INT(2);
    MUL_INT;
    STORE_LOCAL_INT(3);
    PUSH_LOCAL_INT(3);
    STORE_LOCAL_INT(4);
    LOAD_LOCAL_INT(4);
    RET_INT;
return_no_exception:
    RETURN_PRIMITIVE(ret_val);
}

```

As can be seen in the example every byte code is trivially compiled to a macro invocation. All of the above instructions operate on the operand or local variables array, using the fake sp pointer. GCC can optimize this code reasonably well. It is even able to optimize the fake stack pointer away in most cases using aggressive constant propagation and constant folding.

6.2 Dynamically loading the shared library

When a class file has been successfully compiled to C by Jcc, GCC compiles it to a shared library which is then dynamically linked against the running executable. To implement this dynamic linking the “dl” interface originally designed for SUN Solaris is used. The interface declares four functions: `dlopen`, `dlsym`, `dlerror` and `dlclose`. Using `dlopen` a shared library can be opened, returning a shared library handle. Using the handle, the shared library can be searched for symbols given their names using `dldym`. `Dlsym` will return a pointer to the location of the shared object. The shared library can be closed with `dlclose`.

To allow dynamically loaded code to interface with already compiled code, a meta class has information about names, offsets and types of variables and methods. Now consider the following, more complicated example:

```
void start() {  
    System.out.println("Hello World !!!!!!!");  
}
```

Which generates the following byte code:

```
0 getstatic #7 <Field java.io.PrintStream out>  
3 ldc #1 <String "Hello World !!!!!!!">  
5 invokevirtual #8 <Method void println(java.lang.String)>  
8 return
```

Here the global variable “`java.io.PrintStream out`” has to be located and its name passed to the `GET_STATIC` macro. This is implemented by mangling the `out` variable according to the mangling rules of normal Jcc compiled code. Next a string object is created and pushed upon the working stack (operand array). Now matters become complicated. To implement the call, the right function pointer has to be located by indexing the virtual function table of class `PrintStream`. This is implemented by searching the method descriptor table pointed to by the meta class of `java.io.PrintStream`. The meta class itself is located by searching the meta class table. Next the object vtable is extracted from the meta class, and searched for the correct method descriptor. The search is implemented by looking for a matching name and method descriptor string. Method descriptor strings are described in the Java Virtual Machine Specification [10]. The Jcc method descriptor contains the vtable index to call that method. The above code thus compiles to the following C code:

```

int method_test_start0(javaObject *self) {
    int ret_val = 0, flag, sp = 0, call_ret_val = 0;
    TOperandStack operand[MAX_OPERAND_STACK];
    TOperandStack operand[MAX_LOCALS];
    javaObject *call_exception = 0;
    PUSH_STATIC_REFERENCE(_static_java_io_PrintStream_out);
    PUSH_STRING("Hello World !!!!!!!");
    /* call the function over the vtable, function entry 42 (println) */
    call_java_function(((javaObject*) operand[sp-2].reg1)->vtable->entries[42].func,
        2, /* number of arguments */
        &call_exception,
        &call_ret_val);
    sp -= 2;
    return_no_exception:
    RETURN_PRIMITIVE(ret_val);
}

```

7 Jcc Remote Method Invocation

7.1 Java RMI

In Java there is language support for remote procedure calls, RMI in Java speak. The standard implementation however has some drawbacks. For example it is complex to use, has a central name and object server and, most importantly, it is slow. All of these problems are addressed in Jcc-RMI¹. Complexity of RMI has been reduced to a minimum by doing away with the obligatory (magic) interfaces one sees in the standard RMI. Making a class remote and making an instance is implemented in Jcc as follows. First one creates a class and tags it for RMI with the new keyword “remote.” Next, instances of it can be created using the normal language construct “new.” The object can be placed on a specific CPU by using directives such as `RuntimeSystem.setTarget` or `RuntimeSystem.setPolicy`.

When an RMI is executed, instead of calling the method directly a marshal stub is called. It sends the parameters over the network to the unmarshal stub, where the actual call is made. The unmarshal stub next receives the return value or exception object and sends it back over the network to the marshal stub. The marshal stub returns the return value or exception

¹RMI support has been jointly developed by the author and R.V. Nieuwpoort and is more thoroughly described in [11]

Object *	Interface View *
----------	------------------

Figure 26: Interface pointer when local object implementing an interface

object. One detail to be noted here is that when a parameter is an object, and that object contains pointers to other objects, the entire object graph will be sent over the network. For example, to send a linked list over the network, passing the first element of the linked list is sufficient to pass the entire list.

7.2 Compiler support

When executing a “new” statement for a class marked remote, the following steps are made. First a matching constructor is searched for at compile time. When a remote object is to be made, the index in the constructor table for that constructor and the class type-id are send over the network. The other side can then look up the type-id in the class info table to find the meta class for that class. Using the meta class, an object can be created and the right constructor called by indexing the constructor table. Next the object pointer as valid on that machine is sent back. At the receiving end, a remote object stub is wrapped around the object pointer and it is returned.

A method from a remote class can be overwritten by a method from a child class. When an instance of the derived class is created and the method with that prototype is called, the most derived version should be invoked. To implement this behavior, indirection is required formarshallers also. This indirection is implemented using a shadow vtable. In the shadow vtable, every method pointer from the normal vtable is mirrored by an accompanying unmarshal stub pointer. When an RMI is executed, the marshal stub is called for the compile time type of the object the method belongs to. It will marshal the parameters and send them and an index into the shadow vtable to the other side. The other side can then make a virtual function call over the shadow vtable of unmarshallers.

Interfaces pose a special problem for RMI support. Interfaces in Jcc are implemented using fat pointers. An interface pointer is a pair of {vtable for that interface, object pointer}. For a remote class implementing an interface this becomes {vtable ofmarshallers for that interface, remote object stub pointer} (see figure 26 and figure 27). In the remote object stub, at the exact places where the normal object has view pointers, the remote object stub has pointers to tables ofmarshallers for those methods defined in that interface. This makes the casting rules for interfaces as described in section

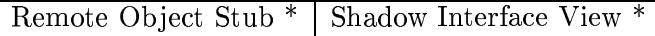


Figure 27: Interface pointer when remote object implementing an interface

3.1 work as expected for remote classes also.

An in depth explanation of how object serialization, buffer management, networking interfaces and more, are implemented in Jcc RMI can be found in [11]. More information about RPC management in general can be found in [18]

8 Garbage collection

8.1 Introduction

Garbage collection in Jcc ² is a fairly standard implementation of mark and sweep. To speed up the garbage collection process, a small amount of compiler support is available. Most importantly this includes the generation of pointer offset tables discussed in section 3.1

In mark and sweep, all objects are registered upon creation. Next when garbage collecting, from the root set (the active stacks and thread objects combined) all reachable objects are visited and marked as reachable (by list traversal). When finished, the mark phase is over and the sweep phase is started. Every object that is not marked reachable is deleted.

There are two major drawbacks with a naive implementation: it would scan far more data than necessary and make multithreaded execution difficult. The first problem is tackled by setting a primitive object bit when the object or array contains no pointers and by adding an offset table to pointers to the vtable of the object. Another factor that greatly helps performance is by enforcing that object pointers are always eight byte aligned. This way, when a word on the stack is checked for being an object pointer only objects with the lower three bits zero need be checked. The second problem is completely dealt with by the runtime system without compiler support. A more extensive description of garbage collection support in Jcc, both local and distributed, can be found in [11].

²Garbage collection support has been jointly developed by the author and J.Maassen and is more thoroughly described in [11]

9 Java exceptions support

9.1 Introduction

Exceptions in Jcc use the “two return value” model of exception handling. Every method thus has two return values, one for the return value and one for the optional exception. In Gasm this is expressed by reserving two or more of the four user visible registers, one for the exception object pointer and one for the actual return value. If a method returns an interface pointer, three return registers must be reserved: two for the interface pointer, and one for the exception object pointer.

If an exception occurs, either by executing a throw statement or by spontaneous events (null pointer exceptions, IO exceptions) an exception object is created and the exception return register is made to point to it. If the exception occurred inside a try statement a jump statement is executed to jump to the first catch statement. Next all catch statements are tried to see if they can accept the exception. A catch statement may accept an exception if the type-id of the catch parameter is declared in the parent table (the parent vtable contains type-ids of all parent classes, including its own type-id). If one catch clause fails to accept the exception, the next one is tried. If there is no catch statement that accepts the exception or the exception has not occurred inside a try statement, a jump statement is executed to jump to the end of the method. The caller next checks the exception return register; if non zero, the same procedure as described above is executed until either the exception is caught or the entire stack has been unwinded. The same exception model has been successfully implemented in Kaffe [22] and Cacao [9].

The two return value model has one major disadvantage, loss of performance. The loss of performance occurs because after every call, the exception return register has to be checked. By inlining small methods we can eliminate the comparison because, in most cases, it will be removed by either jump-to-jump elimination or branch removal by simple constant propagation. An example is displayed in figure 28, before and after inlining. The highlighted code will have become constant because a test is made against a constant value.

9.2 Incorporating Unix signals

In Unix systems, signals give a program information about external events and program failures. For example, if a program erroneously tries to access page zero, a segmentation fault occurs and the program will receive a

```

function foo:
    {do some calculation}
    move R1, 0 {no exception thrown}
end
function main
    call foo
    compare R1, 0
    jump not equal 1:
    {more calculation}
1:
end

```

after inlining:

```

function main
    {do some calculation}
/* deleted by conditional branch elimination:
    move R0, 0
    compare R1, 0
    jump not equal 1:
*/
    {more calculation}
1:
end

```

Figure 28: Optimization of exception code by inlining

<i>void *</i>	<i>start;</i>
<i>void *</i>	<i>catch;</i>

Figure 29: Exception catch structure

SIGSEGV signal. This signal can then be caught by installing a handler function for SIGSEGV. When a signal has occurred, officially no direct information is delivered to the signal handler except the signal number. In most Unix systems a sigcontext structure is pushed onto the working stack before calling the user signal handler. This sigcontext structure contains useful information such as the program counter and stack and base pointer from where the signal occurred. This information is sufficient to allow us to continue execution at the handler for that program counter.

To turn signals into Java exceptions, tables of ExceptionCatch structs are emitted (Figure 29). By default, there is one table entry that spans the entire method. It will have a start field pointing to the start of the method and the catch field pointing to the exit of the procedure. This case is a “catch all” that will direct the exception to the caller of the offending procedure. However, if a method declares a try-catch statement there will be an ExceptionCatch table entry here with the start field pointing to the first statement in the try statement, and the catch field pointing to the first catch clause.

The ExceptionCatch table is sorted at program startup, later when an exception occurs the table can be efficiently searched for a table entry where the program counter falls exactly between the start and catch fields. When such a field has finally been found, a new exception object is created (exact type depends on the signal caught), a setjmp structure is created, the new program counter, exception object pointer and stack and frame pointers are filled in appropriately, and a longjmp is executed.

Kaffe (version 1.0 beta 1) and the Java JDK from SUN also use signals to convert some program failures to Java exceptions. They however have the added complexity and overhead of a signal occurring in the interpreter (Kaffe is a mixed interpreter/JIT and thus may have to deal with this problem also) which must then be turned into a exception for the interpreted program.

9.3 Creating stack traces

In a Java program at any time an exception object may be created and a stack trace be asked of it. A modern processor, when creating a new stackframe, stores the old program counter in it. The old program counter

may, however, point to anywhere in the calling procedure.

Tables of MethodDescriptors are maintained in Jcc for this purpose. The layout of a MethodDescriptor is displayed in figure 16.

When a stack trace is created, all stack frames are visited. From every stack frame, the program counter is extracted. It is then used to search the method descriptor for it, which will immediately deliver the name of the method, method starting address and more. All this information is then passed to the Java code which can then manipulate and print it.

10 Results

All testing is performed on a single Pentium Pro operating at 200 Mhz. The operating system used is BSD/OS 3.0 from BSDI.

Figure 30 shows some small applications tests. Sieve calculates how many times all primes from 0 to 8192 can be calculated in ten seconds. This result is then normalized by calculating the number of seconds it would take to calculate 1000 iterations. This program mainly tests how good the compiler is at tightening a loop. The critical code is about 30 instructions in nested loops. This program also tests array access performance since all prime numbers found are represented by a boolean in an array.

Tsp, the traveling salesman problem, exhaustively tries to find the minimum path along 13 cities. This mainly tests data movement and method call speed.

Ida (iterative deepening A*, [15]), tries to find the minimal number of moves required to solve a puzzle.

Ida mainly tests garbage collector performance. For every move tried a new board is created which is soon thereafter discarded. In total 3.5 gigabytes of object data is created. Kaffe fails to run to completion in this test. Both Kaffe 0.9.2 and version 1.0 were tried.

Figure 31 shows how good Jcc is at optimizing array access and floating point calculations. Linpack benchmark consists mainly of matrix calculations such as matrix inversions, dot-product calculations, scaling etc.

SOR (successive over relaxation) takes a matrix and attempts to smooth it by iteratively decreasing the distance between a value and its neighbors. RayTrace is a simple raytracer for a scene with 2 spheres and a floor. A lot of time is spent in the garbage collector code because many temporary objects are created and immediately thrown away. The JIT and the interpreter are able to perform well using their generational garbage collector.

The results in Figure 33 and Figure 34 show how good Jcc is at optimiz-

ing simple language constructs. All results are in milliseconds required to complete the tests. The better results for the empty loop benchmark from the UCSD benchmark by the JIT can be explained by the slightly different code outputted. For the empty loop, javac outputs loop code with the loop termination test at the end of the loop with an unconditional jump to the test at the start of the loop, improving performance for loops with many iterations. Jcc outputs code to do the loop termination test at the beginning of the loop, increasing performance for smaller loops. That Jcc is better at smaller loops can be seen in Figure 33. The results from the small language constructs benchmark and the UCSD benchmarks should generally not be trusted. Take for example the exception timing benchmark, performance of Jcc seems higher (!) than that of the method call benchmarks while more work is performed (caching effects ?).

We see that the native compiler is in all tests faster than the interpreter and nearly always faster than a JIT by a significant amount. The only exception is object creation where the interpreter and native compiler perform equally well. Object creation is slowed down in the native compiler since it needs to be guarded by a lock and every object created needs to be registered by the garbage collector.

Compiler	Sieve,1000 iterations	Tsp (13) (milli. secs.)	Ida (milliseconds)
Jcc	1.61	3140	78189
Kaffe 0.9.2	2.99	4824	?
Jdk java 1.1	13.88	22998	274159

Figure 30: Small applications

Compiler	LinPack (Mflops)
Jcc	7.5
Kaffe 0.9.2	4.3
Jdk java 1.1	1.1

Figure 31: Linpack, floating point performance test

11 Conclusion and future work

Our work shows that a native off line compiler offers far better performance than an interpreter or JIT compiler without losing much flexibility. Using

Compiler	Radix Sort (100.000 integers)	SOR	RayTrace
Jcc	1996ms	1844 ms	10467ms
Kaffe 0.9.2	2500ms	4789 ms	11032ms
Jdk java 1.1	14488ms	9040 ms	15140ms

Figure 32: Other small applications

Compiler	empty for loop	initializing array of 1000000 ints	object creation
Jcc	16	161	2208
Kaffe 0.9.2	26	270	4040
Java jdk 1.1	206	957	2885

Figure 33: Language constructs timings

a native compiler does potentially lose support for mobile code. As demonstrated, a native compiler and suitable runtime environment can however still support dynamic code loading.

Jcc demonstrates the ease of use of Jade, a parser generator that besides generating the bare parser, also augments the generated parser with code to automatically create the decorated parse tree and maintain scopes. It also generates a lot of utility functions to ease work with the generated scopes and parse trees. We feel that using Jade effectively sliced frontend development time in halve.

We also implemented a new backend that attempts to do some Java specific optimizations, of which some may be applicable to other languages as well. Java specific optimizations include aggressive register promotion (made possible because Java has no pointer arithmetic, generic pointers or variable aliasing in general). Java also makes some optimizations such as copy and constant propagation easier because less analysis has to be performed.

We also present some (larger) changes to the Java native interface to gain performance. The standard native interface (JNI) prescribes that search has to be performed to find addresses and offsets of fields inside objects. In the Jcc native interface, this is no longer needed (but still possible) because the native function can simply use the generated C template for every object pointer. Also, all Java primitive types such as long, int and double have been made compatible with their native C counter parts.

We also added a “remote” keyword to ease programming of RMI. To make a class remote, it now only needs to be tagged as such. There after all calls to a method from a remote class are made using remote procedure

Benchmark	JDK	Jcc	Kaffe
Empty loop iterated 1000000 times:	0.191	0.035	0.022
Added 1000000 ints in loop:	0.338	0.049	0.041
Multiplied 1000000 ints in loop:	0.336	0.053	0.047
Added 1000000 doubles in loop:	0.584	0.189	0.288
Multiplied 1000000 doubles in loop:	1.117	0.700	0.749
Assigned to 1000000 array ints:	0.318	0.060	0.068
1000000 object int field accesses:	0.437	0.035	0.035
1000000 method calls in same object:	0.599	0.071	0.122
1000000 method calls in other object:	0.562	0.070	0.126
Threw and caught 1000000 exceptions:	1.319	0.060	89.362
3 threads, switched 10000 times each:	0.605	0.047	0.512
2000 obj rand. new'd/assigned (avg 2530B), 0 GC'd:	0.110	0.017	0.246
100000 writes of 1 byte:	2.159	1.487	2.773
1 write of 100000 bytes:	0.0080	0.006	0.0070
Cumulative runtime:	8.927	2.997	94.763

Figure 34: UCSD Benchmarks for Java (version 1.1)

calls (RPC). To support efficient RMI, a lot of compiler support had to be provided. Most importantly, marshalling code (in C) and serialization code (also in C) are emitted for every class. This effectively reduces serialization and marshaling costs to a minimum (orders of magnitude faster than the standard RMI implementation from SUN). The exact semantics of the new keyword can be found in [11].

A lot of effort has been spent in optimizing the garbage collection process. It also receives some compiler support by the compiler generating offset tables and generating code that the garbage collector can handle more easily (e.g. 64 bit interface pointers). In the current implementation, garbage collection is about 3-4 times (Ida speedup JDK/Jcc) faster than the garbage collector in the standard Java compiler from SUN and Boehm[8].

Our compiler can be downloaded freely from the web [1] (about 10MB tar file/2.7MB gzipped). The compiler is distributed under the GNU public license (GPL).

A Gasm public interface

Gasm API Index:

OutputProgramTrailer(program_name) and

```

OutputProgramTrailer
FunctionHeader
FunctionTrailer
StartBlock(block-type, one of {LoopBlock, StatementBlock, IfBlock,
SwitchBlock})
EndBlock
CallFunc(name, argument_count_in_words, throws_away_result, float_result)
CallFunc_{R0-R3}(argument_count_in_words)
Load_{R0-R3}(constant)
PopAddress(name, size)
Push_{R0-R3}
Pop_{R0-R3}
PopIndirect_{R0-R3,BP}(offset, size)
PushConstant(constant)
PushLabel(label)
{Divide,Multiply}_{R0-R3}(constant
{Load,Store}Indirect(size, offset)
Move_EBPOff_{R0-R3}
Move_{R0-R3}_EBPOff
Output{Inc,Dec}_{R0-R3}
Shift{Left,Right}_{R0-R3}_{R0-R3}
{Not,BitNot}_{R0-R3}
{Add,Sub,Or,And,ShiftLeft,ShiftRight}_{R0-R3}(constant)
{Add,Sub,Or,And,Xor,Mod,Divide,Move}_{R0-R3}_{R0-R3}
LoadIndirect_{R0-R3}_{R0-R3}(bytes, offset)
LoadAddressLocal_{R0-R3}(offset, size)
LoadAddressParameter_{R0-R3}(offset,size)
Move_EBP_{R0-R3} ————— deprecated
Pop2IntReturnRegister()
Pop2IntReturnRegister2()
Pop2FloatReturnRegister()
PushFloatReturnRegister()
PushDoubleReturnRegister()
PushIntReturnRegister()
PushIntReturnRegister2()
Compare_{R0-R3}(constant)
EmitLabel(label)
Jump{Bigger,Smaller,BiggerEqual,SmallerEqual,Equal,NotEqual}
Jump_{R0-R3}
jump(constant)

```

Push{Float,Double}(constant)
 {Add,Sub,Div,Mul,Mod}{Float,Double}
 Compare{Float,Double}
 Convert{Float,Double,Int}2{Float,Double,Int}
 GlobalAllocDataHeader(name,segment_type,C_style_static)
 AllocGlobalLabel(char*label, segment_type, C_style_static)
 AllocGlobalAddress
 AllocGlobalBytes(size)
 ItemAllocNumber(constant)
 ItemAllocPointer(name)
 ItemAllocSpace(size)
 AddLocalVolatile(offset)
 OutputDebugHeader
 OutputCardinalTypeDebug(name, size, signed, type_id, is_floating)
 OutputFunctionTrailerDebug(name, return-type-debug-id)
 NewAsmFileLocation(line, file)
 AddDebugInfoParameter(name, offset, debugging_id)
 OutputDebugInfoParameters
 DeclareDebugLocalVariable(name, offset, debugging_id)
 {Start, End}DebugBlock
 {Start,End}DefineDebugEnum(enum-name)
 EnterDebugEnumItem(name, constant)

References

- [1] Download site for jcc. <http://www.cs.vu.nl/~rveldema>.
- [2] Java for scientific computing. <http://www.npac.syr.edu/projects/javaforcse/>.
- [3] F.E.J. Kruseman Aretz. On a recursive ascent parser. *Inform. Process. Lett.* vol. 29, no. 4, p201-206, November 1988.
- [4] R. Bhoedjang, T. Ruhl, R. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, September 1993.
- [5] Margaret Ellis and Bjarne Stroustrup. *Annotated C++ Reference Manual*. Addison Wesley Longman, June 1995.

- [6] James Gosling, Guy Steele, and Bill Joy. *The Java language specification*. Addison Wesley, Java series, 1996.
- [7] D. Grune and C. Jacobs. *Parsing Techniques, A Practical Guide*. 1996.
- [8] H.Boehm. Space efficient conservative garbage collection. pages 197–206. ACM SIGPLAN, 1993.
- [9] A. Krall and R. Grafl. CACAO -A 64 bit JavaVM Just-in-Time Compiler. *concurrency: practice and experience*, 1997. <http://www.complang.tuwien.ac.at/andi/>.
- [10] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. SUN press, September 1996.
- [11] J. Maassen and R.V. Nieuwpoort. An efficient distributed runtime for a compiled java environment. Master's thesis, Vrije Universiteit, 1998.
- [12] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufman Publishers, Inc. San Francisco, California, 1997.
- [13] Giles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harissa, a mixed offline compiler and interpreter for dynamic class loading. 1997.
- [14] M. Philippsen and M. Zenger. Javaparty -transparent remote objects in Java. *concurrency: practice and experience*, 1997.
- [15] A. Reinefeld and V. Schnecke. AIDA* -asynchronous parallel IDA*. In *Proceedings 10th Canadian Conference on Artificial Intelligence, AI'94, May 1994, Banff, Canada*. Paderborn Center for Parallel Computing, Germany, 1994.
- [16] George H. Roberts. Recursive ascent: an LR analog to recursive descent. *ACM SIGPLAN Notices, vol 23, no 8, p23-29*, August 1988.
- [17] T. Newsham T.A. Proebsting G. Townsend P. Bridges J.H. Hartman and S.A. Watterson. Toba: Java for applications - a way ahead of time (wat) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, 1997*.
- [18] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.
- [19] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1996.

- [20] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson. Description of a machine architecture for use with block structured languages. Rapport nr ir-81, Vrije Universiteit, Amsterdam, August.
- [21] R. Veldema. Jade, a recursive ascend parser generator for augmented attribute grammars. Technical report, Vrije Universteit, June 1998.
- [22] www.transvirtual.com. Kaffe, a portable mixed jit and interpreter, 1997.

Index

Conversion of SP to FP references,
22

Exception catch structure, 41

Exceptions

two return value model, 39

Field descriptor, 19

Garbage collection

Compiler support for, 38

Gasm, 19

code generation, 19

inline assembly, 29

Jade, 4

Java

features, 27

native methods

calling convention, 27

object

Class, 18

layout, 15

synchronization, 15

parser generator, 4

RMI

Compiler support for, 37

remote method invocation, 36

Unix signals, 39

vtable

in C++, 14

virtual function table, 12